RISC Microprocessor Implementation with Resource Allocation Balanced for Instruction Mix

TR91-035

August, 1991

Manish Pandey

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

RISC Microprocessor Implementation with Resource Allocation Balanced for Instruction Mix

by

Manish Pandey

A thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill, 1991

Approved by:

Akhilesh Tyagi, advisor

Yuki Watanabe, reader

Gyula Magó, reader

Manish Pandey . RISC Microprocessor Implementation with Resource Allocation Balanced for Instruction Mix
(Under the direction of Akhilesh Tyagi.)

Abstract

This thesis explores the *Reduced Instruction Set Computer* (RISC) philosophy the most fundamental principle of which is the *efficient utilizaton of the scarce silicon real estate.* It is conjectured that in keeping with the RISC philosophy one can tailor the datapath to allow each unit in it an area which is justified by the frequency of use of the unit. This would allow one the ability to reallocate the area of different units in a processor to obtain a balanced implementation for a gain in performance.

In an experiment to explore the feasibility of the preceding ideas the integer datapath for the DLX architecture [HP90] is implemented for several design points. The design points implemented include an implementation with a slow (ripple-carry) ALU and a fast (barrel-shifter) shifter. This implementation supports single cycle execution of instructions.

Another design point is implemented with a fast ALU (parallel-prefix) and a slow (linear shift-register) shifter. The extra area taken up by the faster adder is balanced by the savings in area achieved by the slower shifter. In this implementation the cycle time falls even though the CPI increases. The result is that the time per instruction falls when the dynamic instruction mix has far fewer shift instructions than ALU instructions. The implications are that if balancing even a small portion of the chip leads us to a significant performance gain, surely balancing the entire chip gives us even greater opportunities for improved performance.

Acknowledgments

I wish to thank Professor Akhilesh Tyagi for his support, guidance and encouragement, and most of all for his enthusiasm, throughout the course of my research. He left me his door open so that I could discuss problems with him at any time. When things didn't work he offered me encouragement and advice, and this work could never have been completed without his guidance. I also wish to thank Prof. Watanabe and Prof. Magó for their valuable ideas and suggestions in this work.

I must also acknowledge the help I got from Suresh Rajgopal here at UNC, Kris Kozminski at MCNC and Prof Gershom Kedem at Duke University in understanding OASIS, LDVSIM and other CAD tools used to build the processor datapath.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Reduced Instruction Set Computer (RISC) approaches to computer design have come of age and the strengths of the RISC approach are leading to a rapidly increasing market share for RISC architectures at the expense of the so called Complex Instruction Set Computer (CISC) architectures [Ros90],[HP90],[GM87],[KF89], [Tab87]. The RISC approach started as a response to the ever increasing complexity of processor instruction sets which was intended to close the semantic gap between the operations provided in high-level languages (HLLs) and in the machine architectures [Pat85],[Sta90]. However it was discovered that attempts to make instruction set architectures close to HLLs was not the most effective strategy. Instead, compiling programming languages to simple instructions which were most frequently used and making the instruction cycle time as fast as technology would allow, was found to be a better approach [Pat85].

Advances in semiconductor technology have made it possible to fabricate chips containing hundreds of thousands of transistors operating at tens of megahertz frequency [Hen84]. Single chip processors now have a performance comparable to medium to large mainframes of the early eighties. The ever increasing packing density of MOS circuits allows more and more parts of a system to be fitted in a single chip. This helps avoid the speed and cost penalties of having multiple chips in the implementation of a system. Thus, MOS technology has made Very Large Scale Integration (VLSI) an attractive implementation medium for architectures. This introduction of

VLSI has put forth a new problem - that of resource management of both area and time. At any given time the maximum area of a VLSI chip is fixed. This makes the chip area a valuable resource. This brings up the question - What is the best way of allocating the chip area for obtaining maximum performance? The RISC approach is an answer to this question but we feel that current VLSI RISC implementations have still some way to go before the question above can be fully answered [PT91].

1.1 Effective Silicon Utilization

This thesis explores the *Reduced Instruction Set Computer* (RISC) philosophy, [HP90], [GM87], [KF89], [Tab87] the most fundamental principle of which is the *efficient utilizaton of the scarce silicon real estate.* RISC processors today emphasize, among other things, the single cycle execution of instructions ([GM87], [Pat85], [Kat84], [HJP+83],[PGH+84],[Cho89]) in an effort to get a low value of Cycles Per Instruction (CPI)[HP90]. However, there is nothing sacred about single cycle execution of all instructions and this may not necessarily lead to the best possible use of silicon.

To test this hypothesis the datapath for the DLX architecture [described in Hennessy, Patterson [HP90]] is implemented at various design points. The initial design point contains a slow (ripple-carry) ALU and a fast (barrel-shifter) shifter. This implementation supports single cycle execution of all instructions. Available benchmark data for several application programs indicate that the dynamic instruction mix for the DLX processor contains approximately 5% shift class instructions and 35% ALU instructions. So another design point implemented is one where we use a fast ALU (parallel-prefix) and a slow (linear shift-register) shifter. The extra area taken up by the faster adder is balanced by the savings in area achieved by the slower shifter. Even though each shift-class instruction now takes several cycles, the small frequency of the shift class instructions together with the decreased cycle time actually results in a substantial increase in performance.

This demonstrates the feasibility of obtaining performance enhancements when the area allocation is balanced with the instruction mix and points to the need for further investigation of such tradeoffs.

1.2 Contributions of this Work

The new ideas presented in this thesis are the following:

1. The concept of balanced implementation achieveable by design-space exploration of datapath units in a RISC processor.

2. In RISC processor designs, reducing CPI value as close to one as possible should not be the only concern [page1-4,[Ka88]]. Any idea of processor performance is incomplete without the machine cycle time [page 36,[HP90]]. So, the emphasis in RISC processor design should be shifted from reducing CPI to reducing the value of the Average Time Per Instruction (ATPI) , where

$$ATPI = (average\ CPI)(Machine\ Cycle\ Time). \tag{1.1}$$

1.3 Overview of this Thesis

The remainder of the thesis consists of the following four chapters:

1.3.1 Chapter 2 : An Overview and Assessment of RISC

This chapter traces the origins of RISC and then goes on to explore some of the features in contemporary RISC implementations. It ends with a discussion of the manner in which current implementations utilize their chip area.

1.3.2 Chapter 3 : The Design of a RISC Datapath

This chapter describes the decisions made in the selection and implementation of the DLX architecture [HP90] and then describes the more important datapath units implemented.

1.3.3 Chapter 4 : Results and Applications

The area-time measures of the various datapath units in the DLX processor are presented here. The significance of the various measures is explained and the possible applications are mentioned.

1.3.4 Chapter 5 : Conclusion and Further Work

The basic question is how to effectively utilize silicon. Our work has answered only a part of the question. Further work and its possible directions are given here. This chapter also discusses a C compiler, which is a modification of the Gnu C Compiler (GCC), targeted to our processor implementation. This compiler will possibly give us better performance with the DLX processor than the standard GCC compiler.

Chapter 2

An Overview and Assessment of RISC

The concept of RISC is not merely a set of rules dictating the use of few and simple instructions which can be executed by a pipeline that can be implemented efficiently. It goes beyond this. It is a design philosophy dependent on the technology available and the application domain. In the sections that follow we describe the features of current RISC machines. We go on to suggest that further performance gains can be achieved in these designs by adopting a balanced design methodology.

2.1 What is RISC?

Though RISC architectures today emphasize

- instruction sets which are small and simple to decode
- highly optimized pipelines
- single cycle execution of instructions

there is no strict definition of what constitutes a RISC architecture. Rather it is the design philosophy which defines RISC.

2.1.1 The Underlying Philosophy

According to several authors ([Kat84],[Pat85],[GM87]) the design philosophy is basically the one where

1. Target applications are analyzed to determine operations which are most frequent.

2. Those operations which are most frequent are implemented in hardware.

3. An additional instruction/resource is included only if its inclusion does not slow more frequently used operations/resources.

The RISC philosophy espouses freedom to make tradeoffs across boundaries of architecture and implementation, hardware and software, and compile-time and run-time. These tradeoffs can be of different nature depending on the implementation technology, but today with VLSI technology being the technology of choice, most RISC processors have many features in common.We mention some of these features in Section 2.2.

2.2 Common Features of RISC Designs

RISC designs typically have the following features in common [Tab87],[GM87]:

1. Small register-register oriented instruction set with relatively few addressing modes.

2. Fixed instruction formats to facilitate simple hardwired instruction decoding.

3. Instruction set designed for a specific application class.

4. Complex operations are decomposed into several simple instructions.

5. Highly pipelined datapath.

6. Large high speed register file.

7. Hierarchical memory organization with large caches for instruction and data.

8. Single cycle execution of instructions.

9. Heavy dependence on optimizing compilers for performance gains.

An implementation without one or more of the above features is not necessarily non-RISC. What is RISC depends on the specific application domain and the implementation technology used.

2.3 Making Existing RISC Designs More RISCy

As seen in the previous section, single cycle execution of all instructions seems to be a goal of most RISC implementations today [GM87], [HP90], [Kat84], [Cho89], [Ka88], [Pat85]. According to Kane [page 1-4,[Ka88]],

> *The goal of RISC designs is to achieve an execution rate of one machine cycle per instruction.*

This leads to a reduced value of CPI and a lower CPI is indicative of a better performance. But CPI alone does not give one a complete picture of things for it does not include the machine cycle time. This observation is also made in [page337,[HP90]] where two VAX implementations, the 8650 and 8700 are compared. The 8650 has a CPI advantage of 20% over the 8700, but the 8700 has its clock 20% faster than the 8650. The consequence of this is that they both have the same performance [page 36,[HP90]] but it is important to note that 8700 does it with much less hardware. So if a processor design results in increased CPI, it does not automatically follow that its performance will go down. Performance will still improve if the increased CPI is offset by a larger decrease in the machine cycle time.

2.3.1 The Scarce Silicon Real Estate

In [page8,[Kat84]] Katevenis asks the question:

> *Soon, VLSI chips will have significantly more transistors than were used by RISC I or RISC II. What will these additional transistors be used for?*

Designers today use the extra silicon real estate available to add on-chip caches for instructions and data, floating point multipliers and adders, graphics support units etc. [KF89],[Cho89],[Ka88]. The possibilities are enormous but is there a systematic way to utilize the extra area?

Chip area will never be sufficient. No matter what the technology, there will always be yet another subsystem that can benefit from being put on-chip thus creating area shortages. The limiting situation is where we can put an entire computing system on a chip including all the processing units, memory etc. but we are still far from this today. There is a good reason for putting subsystems on-chip - intra-chip communication is much faster and much less bandwidth constrained than communication off-chip. So the silicon resource is indeed a valuable resource and must be judiciously spent.

2.3.2 RISC is Balanced

Since we are interested in high performance, this means that those subsystems which improve performance the most must be allowed to be on the chip. The current RISC trend is a step in the right direction but leaves much to be desired in terms of the tradeoffs between the subsystems which must be present on-chip. The RISC approach applies this analysis across the software-hardware boundary. Why not extend this analysis to hardware design as well? In other words, of the functions to be implemented in hardware, those which are more frequent must be allowed a greater share of chip area. This may be possible, perhaps, at the expense of those functions which are relatively less frequently used and in doing so we do not incur a performance loss [PT91].

We investigate the resource tradeoffs in the design of a datapath of a processor. According to the principle above we should be allocating the area to each datapath unit according to its frequency of use. We term such an implementation a *balanced* one. A balanced implementation is surely a RISC approach for it conforms to the underlying RISC philosophy (Section 2.1.1).

2.4 Problem Statement

This work is an attempt to answer the question *whether it is possible to use a balanced implementation methodology to enhance performance without changing the total area.*

2.4.1 A Balanced Datapath

In this work we investigate the possible tradeoffs between two datapath units in the implementation of a RISC architecture (DLX) described in [HP90]. We try to take a balanced approach to the implementation starting with a conventional implementation and then reallocating area for the two datapath units based on the frequency of their use.

2.4.2 Previous Work

There is no known work in literature addressing the general question of balanced implementation techniques. Kung [Kun86] considered a theoretical model for a computer architecture to study the trade-off between processing rate and I/O bandwidth. Holman and Snyder [HS90b] demonstrate architectural trade-offs in parallel computer design. Our analysis is *budget-constrained* in their terminology. Ho and Snyder [HS90a] give a mathematical formulation of the following principle of balanced design: *The cost of a given part relative to the cost of the entire system must be equal to the time on the critical path spent by that part, relative to the total running time.* Their model, however, fails to consider the balance achievable by exploring the design-spaces of datapath units. In particular, their analysis is limited to the design-space points derived by variation of the *gauge* of an implementation a datapath unit, *i.e.*, ways of realizing 32-bit shift with 4-bit, 8-bit or 16-bit shifter implementations. This research draws on a broader design-space for the datapath unit implementations *e.g.*, for an adder/ALU many schemes such as ripple-carry, carry-select, parallel-prefix, k-bit look-ahead are considered. In addition, this work is more empirical than analytical in nature.

Chapter 3

The Design of a RISC Datapath

This chapter discusses the design decisions made for the implementation of a RISC core datapath for the DLX architecture. It gives a brief overview of the DLX architecture which we selected and the important subsystems in the hardware implementation of the instruction sets.

3.1 Design Decisions

Before we could test the ideas presented in the previous section, we had to have a testbed for exploring them. This meant first deciding on an architecture and then implementing it.

The resources available for this experiment were very limited both in terms of time and manpower. With one graduate student and the time available being less than two semesters, there were severe constraints on the magnitude of the project.

The first option was to work with an already implemented RISC machine [Cho89]. It was rejected because the amount of effort required to first understand the implementation, simulate and measure it and then modify it would have been overwhelming.

So it was decided to first select a 32 bit RISC architecture and then implement it. Again because of the resource constraints we decided to restrict the implementation to only those features on the datapath which were absolutely essential. For the same reason it was decided to rely on automating the design to the fullest extent wherever possible.

3.1.1 Selecting an Architecture

We selected the DLX architecture described in [HP90] because of the following reasons:

1. Public-domain availability of a compiler and a simulator,

2. DLX embodies the essential traits of most contemporary RISC machines.

3.1.2 Selection of Features for Balancing and Performance Gain

Benchmark studies [(with *GCC, SPICE, TEX and US Steel COBOL*)[HP90]] for the DLX indicate that ALU instructions constitute 35% of the dynamic instruction-mix and shift instructions constitute 6% on an average. For any one program from this list the shift frequency is below 5%. This disparity in the relative frequency of the two classes of operations suggests an experiment where the DLX processor can be designed along several design points where we can trade-off the resources required by the ALU and the shifter unit. This can partly answer the question we posed in Section 2.4.

Prior to the experiment there was no available information about the nature of trade-offs between the two hardware units so two extreme design points were chosen for the study. The first design point selected was an implementation containing a slow ripple-carry ALU with a barrel shifter and the other extreme design-point was one with a fast parallel-prefix ALU and a slow linear shifter.

In the second implementation the shifter would take more than one machine cycle to complete the shift operation. There definitely would be a penalty because of the multi-cycle shift operation but this may be minimized by some methods presented in Section 5.2 and in Pandey and Tyagi [PT91].

3.1.3 Selection of an Implementation Methodology

Since this experiment involves building the processor datapath for different design points, we decided to build the datapath elements which were the object of the trade-off study by using OASIS, a standard cell based layout synthesis system [KB88]. To save time, often, random logic in the processor circuit was also implemented with OASIS.

Parts of the design which were regular and could not be automatically synthesized were designed in the full custom methodology using Magic, a layout editing program [SMH+86].

3.1.4 The Influence of MIPS-X on DLX

The similarity of the architectures of DLX and MIPS-X [Cho89] led us to borrow many implementation ideas from the latter (Figure 3.1). This was done to minimize the time redesigning parts for which good designs were already available and to shorten the turn-around time.

The pipeline of the MIPS-X processor is borrowed for DLX in an essentially unchanged form. The strategy for instruction decoding is the same in both except for the hardware implementation. Since the instruction formats for both the processors are different and only the DLX datapath is implemented, the actual decoding logic for both the processors is quite different. Many of the cells used in the MIPS-X datapath were adopted for our design.

We have all along made decisions that allowed us to successfully complete the experiment. Implementing every instruction in the instruction set was not feasible. Also it would not provide us much better results than an implementation of a carefully chosen subset of instructions. The objective of our experiment was to balance the ALU and shift instructions. Therefore, incorporating instructions like multiply, divide and floating point operations would not have contributed much to this objective because these instructions were not a part of the trade-off study. The areas required for implementing these operations would have remained the same across our design points. For this reason, we have chosen not to implement several instructions

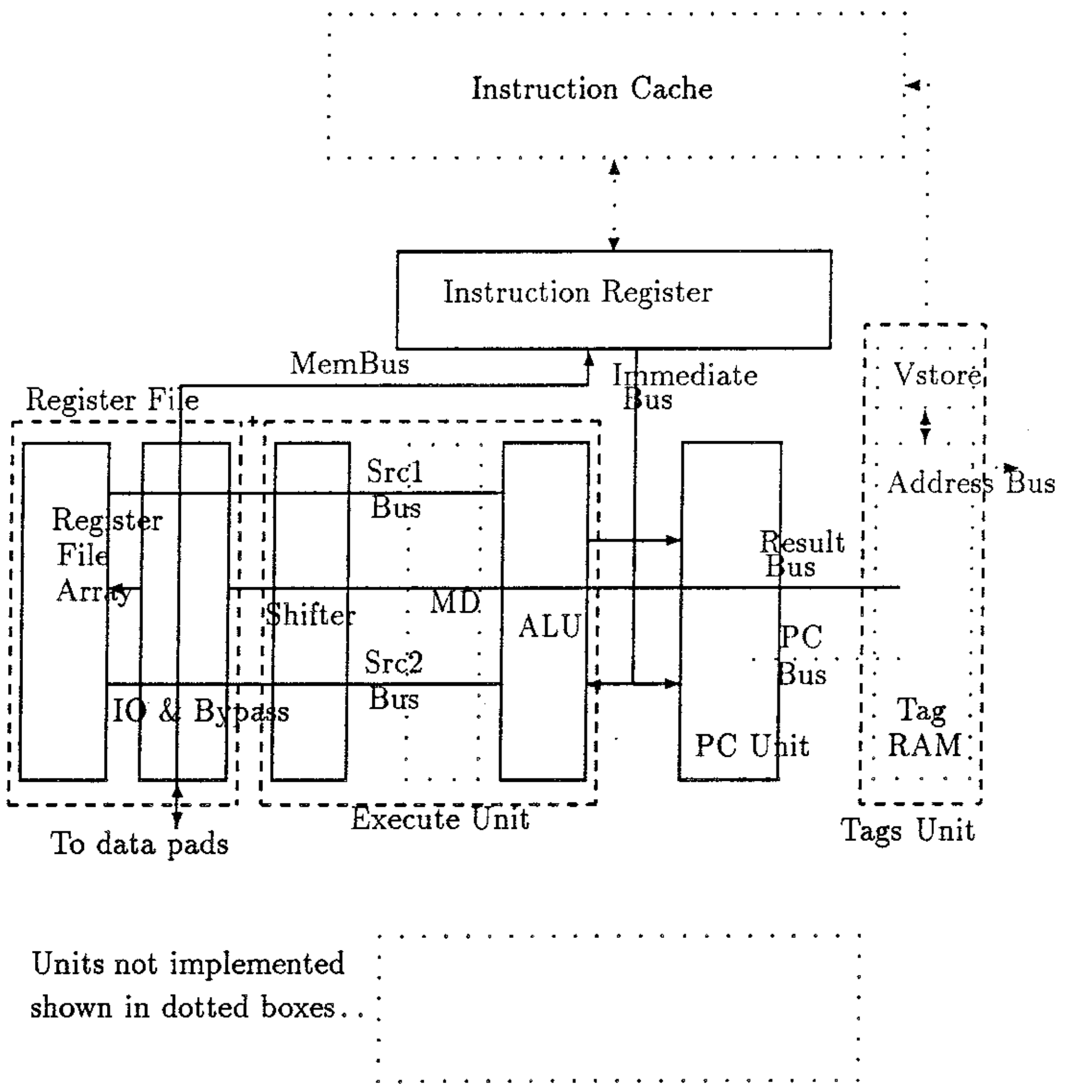


Figure 3.1: The DLX processor: overall organization.

(and their comcomitant hardware), the most notable of which are the multiply and divide instructions. Consequently, we do not have the multiply/divide register in our circuit. In Figure 3.1, the MD register, Tags Unit and the Instruction Cache were not implemented.

Interrupts are not implemented because of the complexity of hardware needed to deal with them and for the same reason we stall the pipeline whenever we encounter a branch instead of having elaborate hardware to minimize the penalty. When the shift operation takes more than one cycle to complete, we again stall the pipeline for the desired number of cycles.

3.2 DLX Overview

3.2.1 The DLX Architecture

This section summarizes some of the more important features of the DLX architecture [HP90]. A complete list of DLX instructions can be found in Appendix A.

- The architecture has thirty-two 32-bit general-purpose registers(GPRs).
- Memory is byte addressable in Big Endian mode with a 32-bit address. All memory references are through loads or stores between the memory and the GPRs.
- All instructions are 32 bits and must be aligned.
- Any GPR may be loaded or stored. The first GPR has 0 hardwired into it.
- There is a single addressing mode, base register plus a 16-bit signed offset.
- All ALU instructions are register-register instructions.
- Control is through a set of jumps and branches. The jumps use a 26-bit signed offset which is added to the program counter.

3.2.2 Implementation Medium for DLX

The technology available to us for designing the datapath was scalable CMOS (MOSIS SCMOS version 7) with two metal layers and a single polysilicon layer. The minimum channel length was $2.0\mu m$.

3.2.3 The DLX Pipeline

The DLX pipeline[1] (adopted from [Cho89]) is 5 stages long (Fig 3.1). The stages in the execution of an instruction are

- Instruction Fetch (IF).
- Register Fetch (RF).
- ALU operation (ALU)
- Memory load/store (MEM).
- Writeback results (WB).

IF_{-1} shown before IF (Figure 3.2) occurs during WB of the previous Stage. The figure gives the details of the operations occuring during each phase. The ALU operation begins during $\phi 1$ of the ALU stage continues after the end of $\phi 1$ but is guaranteed to be complete before the end of $\phi 2$.

3.2.4 Important DLX Subsystems

In the datapath implementations some of the processor subunits have a nontrivial complexity. In this section we discuss the important subunits in some detail. The linear shifter design is discussed in greater detail because of its unusual design.

The implementation of the processor subunits was done in two ways. Highly regular structures or structures which could not be implemented by the suite of standard cells available with OASIS were custom built using Magic, a layout editing program [SMH+86]. This included many basic units on the datapath like latches, tristate

[1]The clocking used by DLX is a two phase non-overlapping scheme.

IF_{-1}	$\phi 1$	No action
	$\phi 2$	PC Bus $\Leftarrow$ displacement adder, trap vector, incrementer, ALU or value from PC Chain
		Precharge tag comparators and valid bit store
IF	$\phi 1$	Do tag compare
		Valid bit store access
		ICache address decoder $\Leftarrow$ PC[bits 26 to 31]
		Detect ICache hit
		Precharge ICache
	$\phi 2$	Do ICache access
		Instruction Register $\Leftarrow$ ICache
RF	$\phi 1$	Do bypass register comparisons to see if bypassing required
	$\phi 2$	Src1 Bus $\Leftarrow$ Src1 register or bypass register
		Src2 Bus $\Leftarrow$ Src2 register, bypass register or offset field in memory instruction
		PC displacement adder latch $\Leftarrow$ branch displacement from Immediate Bus
		Output memory data register $\Leftarrow$ Src2 register or bypass source
ALU	$\phi 1$	Do ALU operation, shifter operation, PC displacement adder addition
		Increment PC (calculate next sequential instruction address)
		Precharge Result Bus
	$\phi 2$	Result Bus $\Leftarrow$ ALU
		Result bypass register $\Leftarrow$ Result Bus
		Memory address pads $\Leftarrow$ Result Bus
MEM	$\phi 1$	No action
	$\phi 2$	Input memory data register $\Leftarrow$ Result register or Memory data pads (load instruction)
		Memory data pads $\Leftarrow$ Output memory data register (store instruction)
WB	$\phi 1$	Destination register $\Leftarrow$ Input memory data register
	$\phi 2$	No action

Figure 3.2: The DLX Pipeline

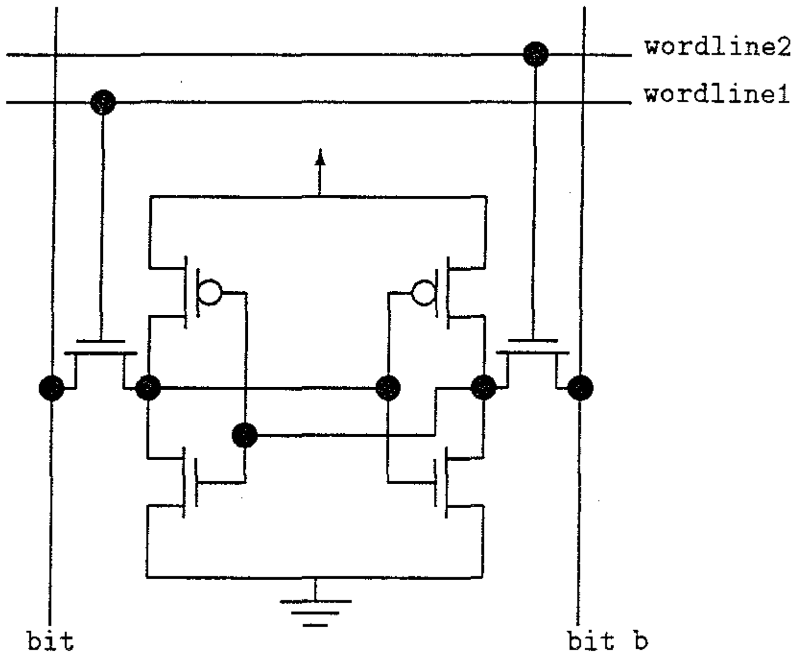


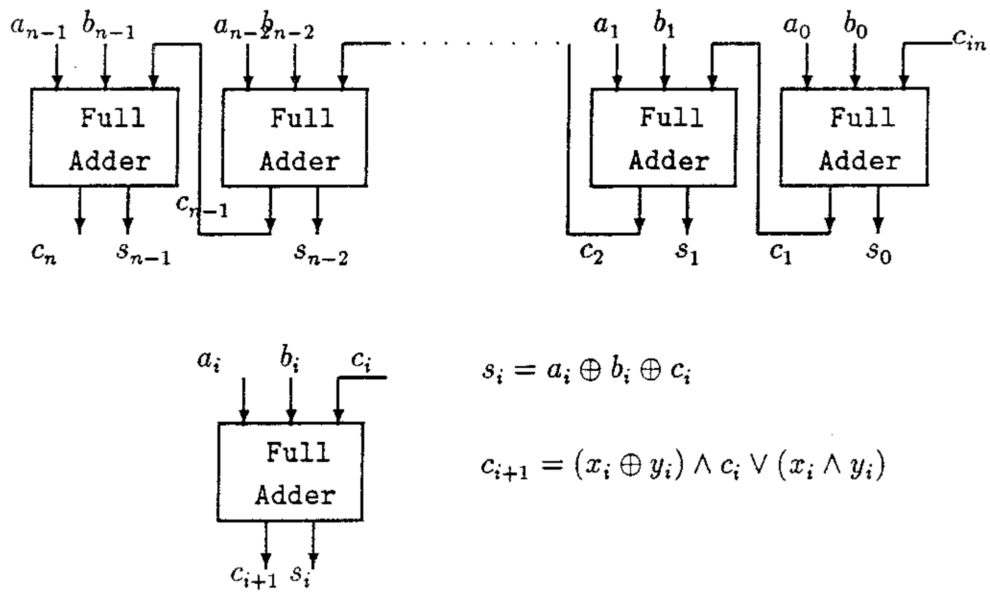
Figure 3.3: The 6-transistor 2-port memory cell used in the register file.

drivers, buffer drivers, memory cells and comparators for bypassing logic in the register file. These units were simulated using CAzM [ERN+89] and were fine tuned to get the best possible performance. Random logic or other combinational circuits which were not very regular were generated using OASIS from their LOGIC-III descriptions.

Register File Unit

The Register File Unit consists of a 32 by 32 register array with decoders, sense amplifiers and drivers, bypassing logic and memory data registers [Cho89].

The register array uses a 6-transistor RAM cell (Figure 3.3) described in [SKP+84]. The cell layout was done using Magic and was tested with CaZM [ERN+89]. Because of the small size of the array, sense amplifiers were not necessary and just an inverter sufficed to detect the signal changes in the array *bit* and *bit_b* (complement of bit) lines [page52, [Cho89]].



$$s_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = (x_i \oplus y_i) \wedge c_i \vee (x_i \wedge y_i)$$

Figure 3.4: Ripple Carry Adder for n bits.

ALU

The ALU was implemented primarily using OASIS. LOGIC-III descriptions [KB88] of the various subunits, notably the adder-carry structure and the boolean function unit, were compiled. The tristate drivers [page 46,47 [Cho89]] for driving the buses were custom designed and tested. Then they were combined with the standard cell layouts generated by OASIS in order to form the complete ALU unit. Another approach tried was to include the tristate drivers as standard cells, in OASIS and then use them for generating the complete ALU unit. Because of the problems with characterizing non-combinational cells and the stringent layout requirements for OASIS standard cells, the latter approach was abandoned.

The adder is the basic unit around which other arithmetic functions like subtraction etc. are built. We implemented two flavours of adders: ripple carry (Figure 3.4) and parallel prefix (Figure 3.5) [HP90]. The parallel prefix adder is referred to as a complete carry-lookahead tree adder in Hennessy and Patterson [page A-35 [HP90]].

The design points implemented contain a 32-bit ripple carry adder, a 32-bit parallel prefix adder and four 8-bit parallel prefix adders joined end to end in ripple carry

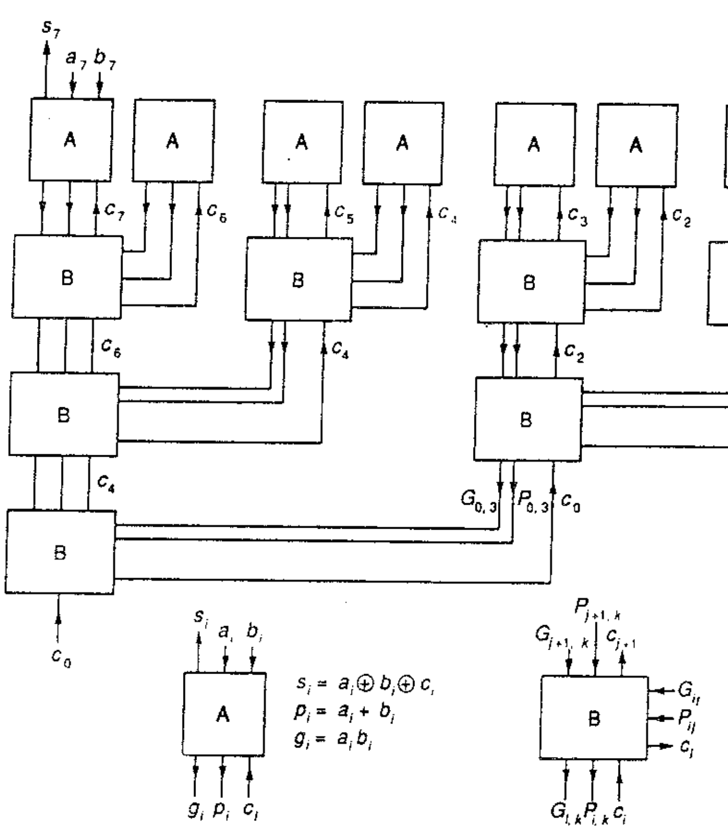


Figure 3.5: Parallel Prefix Adder for 8 bits.

fashion(Figure 3.6). Appendix A.1 contains the LOGIC-III description of a ripple carry adder and A.2 contains the LOGIC-III description of a parallel prefix adder. OASIS was used to generate the standard cell layouts from the LOGIC-III descriptions for the modules mentioned above.

The boolean function unit implements the operations logical OR, logical AND and logical XOR. OASIS was used to generate this unit.

Shifter

In this experiment we include two types of shifters in our designs, namely a barrel shifter and a linear shift register. The barrel shifter (Figure 3.7) is described in LOGIC-III in Appendix B.3. A standard cell layout for this shifter was generated using OASIS.

The barrel shifter completes the shift operation in one machine cycle time. The linear shifter takes more than one machine cycle time because a shift by amount n is done with n linear shifter shifts which takes n clock time periods. Simulations

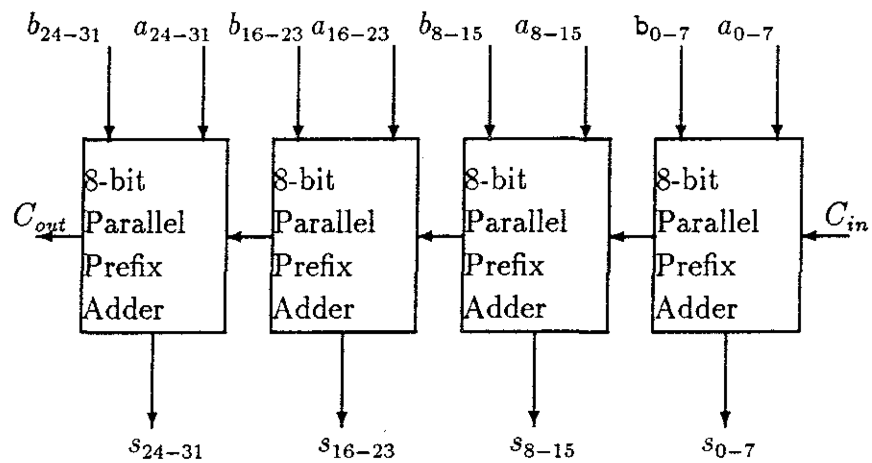


Figure 3.6: 32-bit Adder using 8-bit Parallel Prefix Adders.

indicated that the linear shifter could function much faster than the global machine clock. So it was decided that the linear shifter would be clocked at the maximum possible speed at which it could run. This was done so that a shift would take the minimum number of machine cycles. This minimizes the performance penalty due to making the shift operation a multicycle operation.

Conceptually the simplest way to run the linear shifter at a rate faster than the global machine clock would have been to have another faster clock externally input to the processor. But this approach was rejected because of problems like clock distribution, clock synchronization and extra pins required for this additional input. Instead, it was decided to have an internal source of clock signals to drive the linear shifter. This was achieved using self timed circuits to generate the necessary signals.

The linear shifter is a self timed circuit. It was implemented in the full-custom methodology. Figure 3.8 shows a part of the circuit. The lower part of the figure shows a chain of inverters made up of transistors q_1 through q_{10}. Since there are an odd number of inverters in the chain, there exists a possibility of oscillation when the output of the rightmost inverter can reach the input of the leftmost inverter. This is possible when the signal *go* is high. A counter stores the value of the shift amount in a shift instruction and decrements it with each shift. The signal *go* remains high as

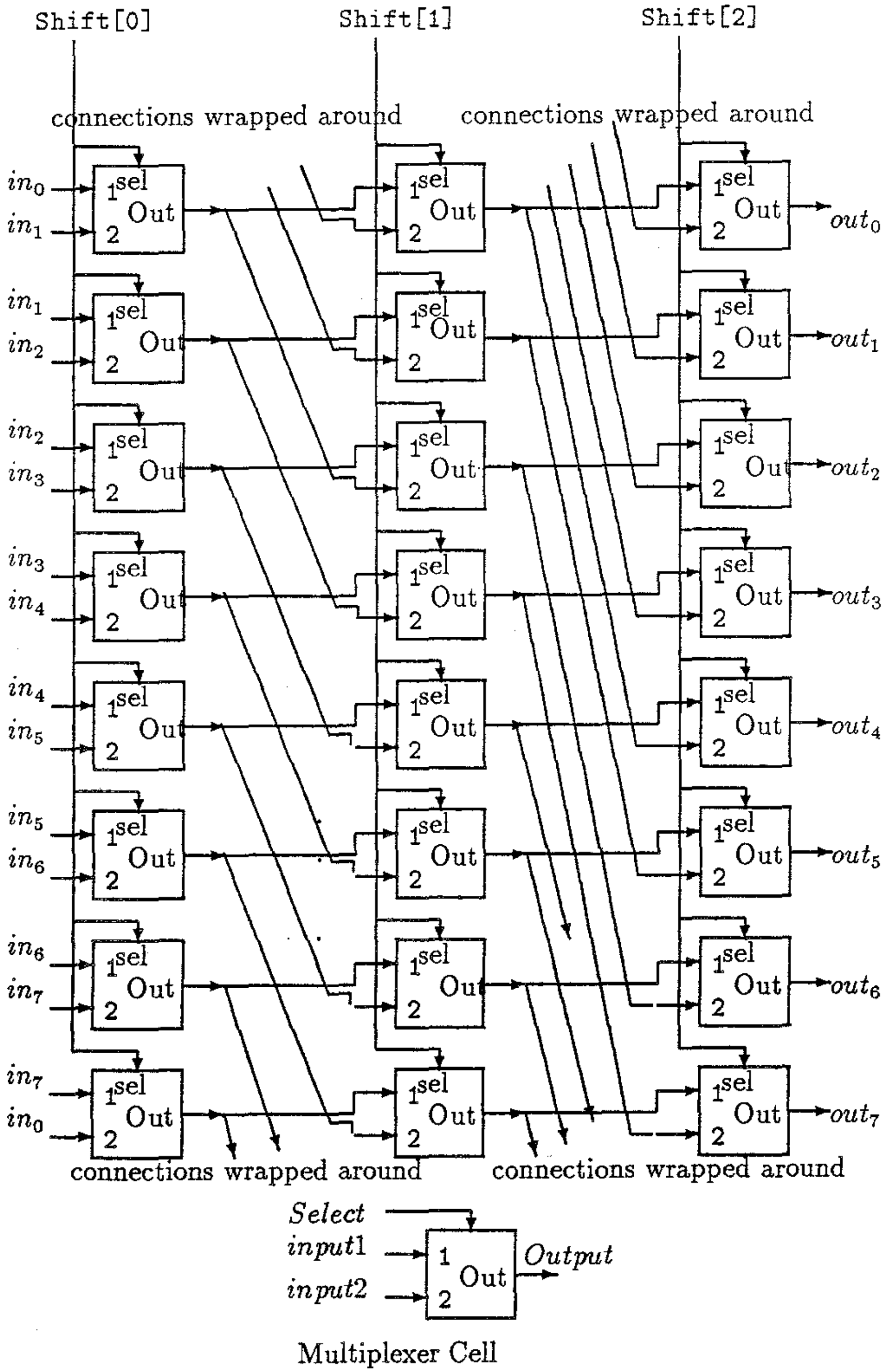


Figure 3.7: 8-bit Barrel Shifter.

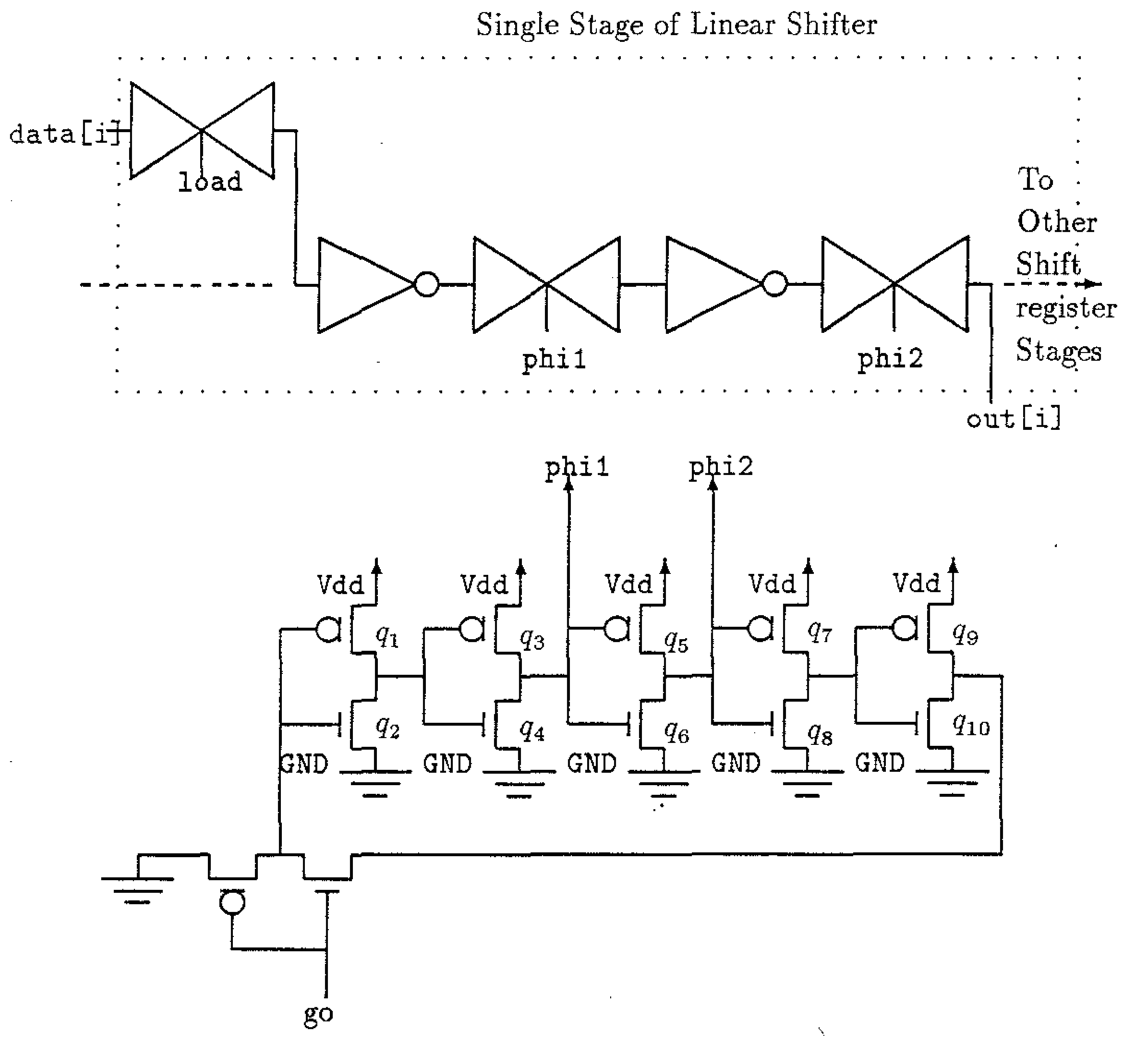


Figure 3.8: Shift register circuit including a ring oscillator for generating shift signals.

long as the counter value is non-zero.

The W/L ratios of transistors q_7 through q_{10} in Figure 3.8 have been chosen carefully so as to provide the desired frequency of oscillation (which allows reliable shifting). Transistors q_3 through q_6 have exceptionally large W/L ratios[2] to allow them to drive phi1 and phi2 of all the 32 shift register stages [WE85].

[2] Width/Length ratios q_3:40, q_4:80, q_5:25, q_6:50.

Program Counter

The Program Counter (PC) closely follows the organization given in [Cho89]. There is an incrementer for incrementing the value of the PC to the next sequential value. Also there is a displacement adder for adding to the PC branch displacements etc.. We have used two adders in the PC unit. These adders are similar to the one in the ALU because the time required for generating a new address by the PC unit or the time for computing a new result by the ALU must both satisfy similar timing requirements.

Instruction Register and Decode Circuitry

The important parts of this unit are a register array and a decoder array [page151-158, [Cho89]]. The register array is implemented as a shift register of two or three stages for each bit in the instruction. This array serves as the input to the decoder array which generates the control signals for various units. The decoder array is implemented in standard cells with the help of OASIS.

3.3 Implementing DLX at Various Design Points

The first design point to be implemented was one where all instructions have uniform cycle length. This implementation contained a ripple carry adder in the ALU and a barrel shifter in the shifter unit. The PC adders were two ripple carry adders.

The second design point implemented was one with a parallel prefix adder in the ALU and a linear shifter. The PC unit, as above, contained two parallel prefix adders.

The third implementation used 4 cascaded 8-bit parallel prefix adders (Figure 3.6) in the ALU and PC units and retained the linear shifter.

Because of the use of a barrel shifter, each shift operation in the first implementation took only one cycle. This implementation followed the *one instruction per cycle* paradigm followed by most RISC implementations today [GM87].

The second and third implementations took more than one cycle for shift amounts greater than one. The reason for this is that a shift by amount n is done by n successive shifts on the linear shifter (Figure 3.8). In these implementations, the pipeline was stalled till the shift operation was completed.

Chapter 4

Results and Applications

In this chapter area-time measures of the various DLX implementations are given and the significance of the various measures is explained. The possible applications of this work are mentioned in the last section.

4.1 Area-Time Measurements

The results of the simulations carried out to determine the worst case processor cycle time are given in Section 4.1.1 . The area of the layouts of the various units can be found in Section 4.1.2 .

4.1.1 Simulation of DLX

Several simulation tools were used for testing the various units individually and then the complete system.

The Register File (RF) unit was simulated completely using CAzM [ERN+89]. This was done because switch level simulation tools failed to accurately simulate the 6-transistor memory cell. However, the complete 32 by 32 RF was not simulated using CAzM at the same time because of the very large amount of simulation time required. Instead, one complete row together with one complete column in the 32 by 32 register file was selected and simulated. This was done because involving a complete row would accurately model the total capacitances on the bit and bit_b lines

unit	delay (ns)
ALU with Ripple Carry Adder	93
ALU with Parallel Prefix Adder (PPA)	33
ALU with 4 8-bit PPA	66
Barrel Shifter	27
Linear Shifter (worst case time)	320

Table 4.1: Delay Figures for ALU and Shifter unit.

(Figure 3.3) [page 52, [Cho89]] and including the column would accurately reflect the total wordline delays [page 53, [Cho89]]. It was found that the worst case delays were 10 ns to either read or write from the register file unit alone.

However, all control signals are generated by the controller which has a worst case delay of 4 ns and then the signals are distributed by large AND drivers [page 48, [Cho89]] which have a worst case delay of 6 ns. Also if some functional unit puts some data on a bus there may be a maximum delay of upto 3 ns.

So the register file mentioned earlier actually has a read/write time of 10+4+6 ns i.e. 20 ns when it is integrated into the complete system.

The ALU and Shifter units were tested with LDVSIM [Bri89] for functionality and then switch level simulation was done using RNL [L.I87] to determine their delays. The results of their simulation are presented in Table 4.1. The delay figures in the table include all the delays including the controller delay, buffer delay and time taken for the result to be written on the bus. The program counter unit is faster than the ALU by 3 ns and it is not on the critical path.

4.1.2 Layout

The first design point implementation with a barrel shifter and a ripple carry adder in shown in Figure 4.1. The area figures for the various datapath units in this implementation are given in Table 4.2. The datapath areas with a linear shifter and various adders are in Tables 4.3 and 4.4.

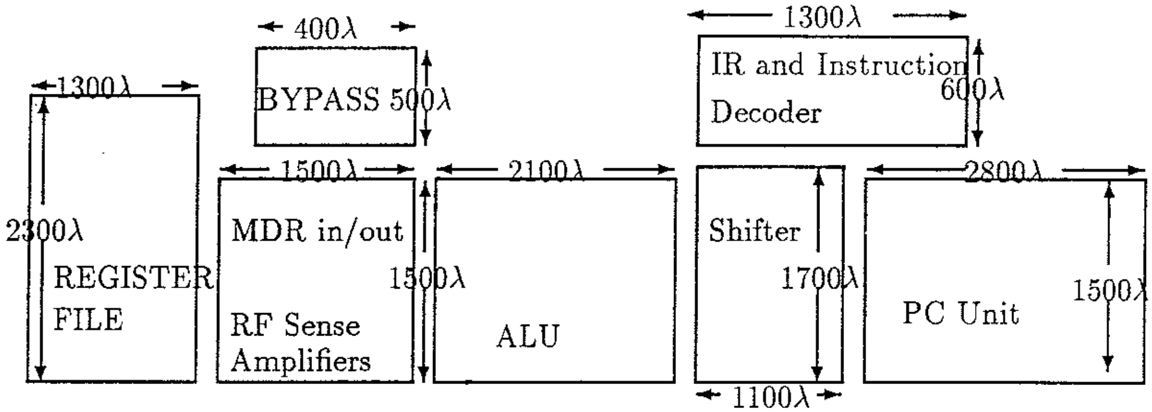


Figure 4.1: The Datapath using a Ripple Carry Adder and a Barrel Shifter.

unit	area (λ^2)
Register File	3×10^6
32-bit Ripple Carry ALU	3.2×10^6
Barrel Shifter	1.9×10^6
PC with 32-bit Ripple Carry Adder	4.2×10^6

Table 4.2: Area Figures for the Datapath with Barrel-Shifter (First Implementation).

unit	area (λ^2)
Register File	3×10^6
Shifter	0.2×10^6
ALU with 32-bit Parallel Prefix Adder	3.8×10^6
PC with 32-bit Parallel Prefix Adder	5.4×10^6

Table 4.3: Area Figures for the Datapath with a 32-bit Parallel Prefix Adder (Second Implementation).

unit	area (λ^2)
Register File	3×10^6
Shifter	0.2×10^6
ALU with 4 8-bit Parallel Prefix Adder	3.3×10^6
PC with 4 8-bit Parallel Prefix Adder	4.4×10^6

Table 4.4: Area Figures for the Datapath with 4 8-bit Parallel Prefix Adders (Third Implementation).

4.2 Comparison of Various Implementations

According to our simulations the ALU operation is in the critial path of the datapath operation. Therefore, the speed of ALU operation, including writing result on to the bus, determines the cycle time for the datapath. From Table 4.1 it is clear that the cycle times for the three implementations are 93ns, 33ns and 66ns respectively. In the latter two implementations we have a linear shifter which takes 320ns to complete its operation in the worst case. So it is clear that in the second implementation shift takes 9 cycles (320ns/33ns - 1) more than an implementation which completes a shift instruction in 1 cycle. In the third implementation the shift instruction takes 4 cycles more than an implementation which completes a shift instruction in 1 cycle.

The CPI figure for the DLX processor with single-cycle instructions only is 1.42 [page 277, [HP90]]. With shifts taking 9 extra cycles the new value of CPI is $1.0 \times 1.42 + .05 \times 9$ which equals 1.87. With shifts taking 4 extra cycles the CPI value turns out to be $1.0 \times 1.42 + .05 \times 4$ which is 1.62.

The CPI value for the three implementations in order are 1.42, 1.87 and 1.62. However, because the cycle time for the second processor is smaller than the others, it is the fastest design with a time per instruction value of $1.87 \times 33ns$ which equals 61.7ns. Table 4.5 summarizes our findings.

Table 4.5 clearly shows that even with a higher value of CPI we can obtain better performance if the machine cycle time is small enough. Furthermore, the area taken by the faster design can be the same as the area occupied by the slower design (First

design	cycle time (ns)	CPI	time per instruction (ns)	datapath area (λ^2)
original	93	1.42	132	9.3×10^6
second	33	1.87	61.7	9.4×10^6
third	66	1.62	106.9	7.9×10^6

Table 4.5: CPI, Time per Instruction and Area Comparison of Three Implementations

and Second designs) or even less (First and Third designs).

4.3 Applications

The instruction mix skew between the ALU and Shift class instructions enabled us to redesign our datapath to obtain better performance within the same area. This indicates the possibility of targeting application areas which exhibit significant instruction-mix skew. Two such areas are the X terminal and the embedded controller.

4.3.1 X Terminals

X Terminals are primarily used to run the X Windows software. The tasks involved are providing a graphic window interface and network communications both of which are likely to have a large instruction-mix skew.

The instruction mix of network communication programs [1] as reported in Smith [Smi78] is highly skewed. Shifts account for less than 1% of the total instruction mix whereas 37.85% of the instructions are branch instructions and Load/Store instructions constitute another 36.68% of the instruction mix.

This gives an idea of the nature of the instruction mix skew arising as a result of the X terminal executing network communication software and suggests the kind of trade-offs which may be necessary to tailor a RISC processor for such a task. It

[1] For an IBM370/155 running only network communication programs.

points to the possible tradeoffs between specialized branch prediction hardware, larger register file and cache and a reduced shifter and arithmetic unit.

4.3.2 Embedded Controller

Many microprocessors are used as a part of a control system rather than the CPU of a general purpose computer. They are used in diverse environments: from desktop peripherals like printers [Wir91] and scanners to audio-visual equipment and factory automation machinery to automobiles. Typically these microprocessors execute very small number of programs in their life time. The dynamic instruction mix of these applications can have potentially a large skew as compared to the instruction mix of a set of typical application programs run on a UNIX workstation. Therefore there is a large potential for an improved processor design based on these instruction mixes. With the introduction of RISC processors for embedded control applications [Tho90] this work becomes more important as balanced designs can offer an area-constrained design (with faster time) or a time-constrained design (with reduced area) for the same task.

RISC is expected to successfully penetrate the computer peripheral portion of the embedded control market [Ros90] where performance has higher priority over price. Already RISC microprocessors are being used for near-real-time applications [Wei91] like laser printer control [Wir91], graphics and data staging. Since these tasks are very specialized, the RISC processor balanced for the instruction skew can potentially offer a better performance. This is an application of area-constrained balancing, where improved performance is the point of interest.

Chapter 5

Conclusion and Further Work

This chapter summarizes the lessons learned from the experiment carried out and points to its implications. It concludes with the work in progress and gives directions for future work .

5.1 Implementations Balanced for Instruction-Mix Skew

Our starting point was based on the premise : *Silicon is an expensive resource and hence it should be allocated to only those functions that can justify it by the frequency of their use.* Starting from this principle, we studied the instruction mix skew of two instruction classes for some general purpose programs typically run on a UNIX machine. We then designed a RISC processor datapath which 'violated' the RISC principle by having a multicycle shift operation. This design was motivated by the instruction mix skew mentioned earlier . But this datapath at the same time had a better performance than the one with uniform-length instruction execution times. *It seems that RISC processor designers have fallen into the rut of uniform-length instruction execution times.* The experiments we have conducted demonstrate that there is nothing sacrosanct about uniform-length instruction execution times. If our instruction-mix skew information so indicates, then we should be free to do trade-offs between the hardware units so that the resource allocated to each is justified by the frequency of use of the unit. Such trade-offs lead to a balanced design with a potential gain in performance over processors without such balancing and these trade-offs are in keeping with the RISC spirit.

Our experiments have demonstrated the feasibility of performance gain when two small datapath elements were redesigned and the design was area constrained. In RISC microprocessors [Per89] there are many other components like on-chip caches, floating point units, register files etc.. Redesigning these components so that the resources they take (area) match their frequency of use will potentially free up areas from some components which can be used by other components. In such an area constrained redesign there exists the possibility of obtaining performance gains beyond that obtainable by balancing only the datapath.

5.2 Extensions to this Work

5.2.1 Compiler

We are currently working on the GCC compiler for DLX [Sta89]. We intend to modify it for running it on the next version of the DLX processor under design. Currently, the multicycle shift instruction involves stalling the pipeline for the duration of the shift operation. During this phase no other instruction is fetched and the other functional units remain idle. An alternative to this would be to give the shift operation a fixed number of cycles to complete and while shifting is being done we can fetch independent instructions and execute them, thus achieving a better utilization of hardware resources.

Towards this end we are in the process of modifying the GCC compiler. The first stage in the modification has been completed where a fixed number of NOP slots have been appended after each operation of the shift class. This has been done by a modification of the machine description files for the DLX machine.

The next stage is to analyze the code to find out independent machine instructions which can fill up the empty slots after the shift instructions. This work is under progress.

5.2.2 Further Work on RISC Core Datapath

Since in our experiment the register file was not on the critical path, it was not included in the trade-off studies done. However, the size and organization of the register file has a significant impact on the number of load/store instructions executed by a program.

The register file versus load/store frequency follows the *knee-curve* behaviour[pages 450-451, [HP90]]. When an instruction exhibits a high percentage of load/store instructions (leading to a higher CPI), a natural question to ask is whether the register file size is below the knee of the curve and more area needs to be allocated to it. Similarly, the register file may be overdesigned (size is considerably higher than the knee). Under such circumstances a reallocation of register file area to other components may lead to a better performance. Further experiments need to be devised to study this aspect of the size of the register file vs other components based on the instruction mix.

5.2.3 Caches

The dependence of miss rates on the cache size follows a knee curve [HP90]. We need to quantize this relationship with respect to cache size vs cycle time/CPI trade-off. How the choice of designs for sense-amplifiers, tag-comparators and line drivers (and the areas required by the different designs) affects cache performance is not known quantitatively. We need to be able to independently manipulate cache size and cache circuit design to vary the CPI and the cycle time. Building such models will help us better understand the answer to the underlying question of when to reallocate some area from/to cache to reduce/increase CPI.

5.2.4 Control

As the processor is balanced, there are more and more instructions which can have non-uniform cycle length. This may lead to a more complex control unit which then takes up more area. At the turning point, the benefits of balancing the processor will be outweighed by the increase in area and delay of the complex control unit. Where this point occurs is important to know for a given set of instructions.

5.2.5 Multiply/Divide and Floating Point Hardware.

In many application domains, to do any useful job the processor must have the hardware for multiply/divide and it must also perform floating point operations. Extending the idea of balanced design here is needed to study the performance issues depending on the amount of area resource given to each functional unit and the instruction mix.

5.2.6 Balanced Design for Specialized Application Domain.

After the issues in Sections 5.2.1 to 5.2.5 are more clearly understood, a validation of our ideas would require the complete design of a processor. This processor would be designed for a specific application domain like the X terminal processor using balanced design techniques (taking into account the instruction-mix skew). We would then compare this balanced processor with a straightforward implementation of the same processor (with each instruction taking uniform time to execute) similar to our experiments described in Chapter 3.

Appendix A

The DLX architecture

A.1 The DLX Instruction Set

The complete instruction set for DLX appears in the next page.

A.2 DLX Instruction Formats

The figure containing the DLX instruction formats appears after the next page.

All DLX instructiona are 32 bits with a 6-bit primary opcode.

A.3 Bit Assignment for Instructions

In the opcode tables below the first column contains the mnemonic opcode. The second column indicates the instruction format. RFMT refers to the format of the register-register instructions. (See Appendix A.2) JFMT refers to the format of the jump instructions and IFMT is the format of the instructions with am immediate operand. The third column contains the bit assignments for the instructions. Here "rrrrr" or "RRRRR" refer to the 5-bit register address. "ooo...ooo" is the 26-bit offset in the jump instructions and "iiiiiiiiiiiiiii" is the 16-bit immediate quantity. "x..x" refers to dont cares. These tables have been decoded from the software available with [HP90].

Instruction type / opcode	Instruction meaning
Data transfers	**Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR**
LB, LBU, SB	Load byte, load byte unsigned, store byte
LH, LHU, SH	Load halfword, load halfword unsigned, store halfword
LW, SW	Load word, store word (to/from integer registers)
LF, LD, SF, SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, MOVS2I	Move from/to GPR to/from a special register
MOVF, MOVD	Copy one floating-point register or a DP pair to another register or pair
MOVFP2I, MOVI2FP	Move 32 bits from/to FP registers to/from integer registers
Arithmetic / Logical	**Operations on integer or logical data in GPRs; signed arithmetics trap on overflow**
ADD, ADDI, ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT, MULTU, DIV, DIVU	Multiply and divide, signed and unsigned; operands must be floating-point registers; all operations take and yield 32-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate—loads upper half of register with immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts: both immediate (S__I) and variable form (S__); shifts are shift left logical, right logical, right arithmetic
S__, S__I	Set conditional: "__" may be LT, GT, LE, GE, EQ, NE
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ, BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
BFPT, BFPF	Test comparison bit in the FP status register and branch; 16-bit offset from PC+4
J, JR	Jumps: 26-bit offset from PC (J) or target in register (JR)
JAL, JALR	Jump and link: save PC+4 to R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address; see Chapter 5
RFE	Return to user code from an exception; restore user mode; see Chapter 5
Floating point	**Floating-point operations on DP and SP formats**
ADDD, ADDF	Add DP, SP numbers
SUBD, SUBF	Subtract DP, SP numbers
MULTD, MULTF	Multiply DP, SP floating point
DIVD, DIVF	Divide DP, SP floating point
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Convert instructions: CVTx2y converts from type x to type y, where x and y are one of I (Integer), D (Double precision), or F (Single precision). Both operands are in the FP registers
__D, __F	DP and SP compares: "__" may be LT, GT, LE, GE, EQ, NE; sets comparison bit in FP status register

Figure A.1: Complete List of instructions in DLX. [HP90]

I - type instruction

6	5	5	16
Opcode	rs1	rd	Immediate

Encodes: Loads and stores of bytes, words, half-words
All immediates (rd ← rs1 op immediate)

Conditional branch instructions (rs1 is register, rd unused)
Jump register, Jump and link register
(rd = 0, rs = destination, immediate = 0)

R - type instruction

6	5	5	5	11
Opcode	rs1	rs2	rd	func

Register–register ALU operations: rd ← rs1 func rs2
Function encodes the data path operation: Add, Sub , . . .
Read/write special registers and moves

J - type instruction

6	26
Opcode	Offset added to PC

Jump and jump and link
Trap and RFE

Figure A.2: Instruction layout for DLX. [HP90]

OPCODE TABLES FOR DLX

OPCODE LIST 1: Primary Opcodes

```
OPCODE   FORMAT     BIT ASSIGNMENT FOR MACHINE INSTRUCTION
------   ------     --------------------------------------


SPECIAL  RFMT      0000 00rr rrrR RRRR rrrr rxxx xxvv vvvv
                                        -- see OPCODE LIST 2
FPARITH  RFMT      0000 01rr rrrR RRRR rrrr rxxx xxvv vvvv
                                        -- see OPCODE LIST 3
J        JFMT      0000 10oo oooo oooo oooo oooo oooo oooo
JAL      JFMT      0000 11oo oooo oooo oooo oooo oooo oooo

BEQZ     IFMT      0001 00rr rrrR RRRR iiii iiii iiii iiii
BNEZ     IFMT      0001 01rr rrrR RRRR iiii iiii iiii iiii
BFPT     IFMT      0001 10rr rrrR RRRR iiii iiii iiii iiii
BFPF     IFMT      0001 11rr rrrR RRRR iiii iiii iiii iiii

ADDI     IFMT      0010 00rr rrrR RRRR iiii iiii iiii iiii
ADDUI    IFMT      0010 01rr rrrR RRRR iiii iiii iiii iiii
SUBI     IFMT      0010 10rr rrrR RRRR iiii iiii iiii iiii
SUBUI    IFMT      0010 11rr rrrR RRRR iiii iiii iiii iiii

ANDI     IFMT      0011 00rr rrrR RRRR iiii iiii iiii iiii
ORI      IFMT      0011 01rr rrrR RRRR iiii iiii iiii iiii
```

```
XORI    IFMT    0011 10rr rrrR RRRR iiii iiii iiii iiii
LHI     IFMT    0011 11rr rrrR RRRR iiii iiii iiii iiii


RFE     IFMT    0100 00rr rrrR RRRR iiii iiii iiii iiii
TRAP    IFMT    0100 01rr rrrR RRRR iiii iiii iiii iiii
JR      IFMT    0100 10rr rrrR RRRR iiii iiii iiii iiii
JALR    IFMT    0100 11rr rrrR RRRR iiii iiii iiii iiii

RES     IFMT    0101 00rr rrrR RRRR iiii iiii iiii iiii
RES     IFMT    0101 01rr rrrR RRRR iiii iiii iiii iiii
RES     IFMT    0101 10rr rrrR RRRR iiii iiii iiii iiii
RES     IFMT    0101 11rr rrrR RRRR iiii iiii iiii iiii

SEQI    IFMT    0110 00rr rrrR RRRR iiii iiii iiii iiii
SNEI    IFMT    0110 01rr rrrR RRRR iiii iiii iiii iiii
SLTI    IFMT    0110 10rr rrrR RRRR iiii iiii iiii iiii
SGTI    IFMT    0110 11rr rrrR RRRR iiii iiii iiii iiii

SLEI    IFMT    0111 00rr rrrR RRRR iiii iiii iiii iiii
SGEI    IFMT    0111 01rr rrrR RRRR iiii iiii iiii iiii
RES     IFMT    0111 10rr rrrR RRRR iiii iiii iiii iiii
RES     IFMT    0111 11rr rrrR RRRR iiii iiii iiii iiii


LB      IFMT    1000 00rr rrrR RRRR iiii iiii iiii iiii
LH      IFMT    1000 01rr rrrR RRRR iiii iiii iiii iiii
RES     IFMT    1000 10rr rrrR RRRR iiii iiii iiii iiii
```

LW	IFMT	1000 11rr rrrR RRRR iiii iiii iiii iiii
LBU	IFMT	1001 00rr rrrR RRRR iiii iiii iiii iiii
LHU	IFMT	1001 01rr rrrR RRRR iiii iiii iiii iiii
LF	IFMT	1001 10rr rrrR RRRR iiii iiii iiii iiii
LD	IFMT	1001 11rr rrrR RRRR iiii iiii iiii iiii
SB	IFMT	1010 00rr rrrR RRRR iiii iiii iiii iiii
SH	IFMT	1010 01rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1010 10rr rrrR RRRR iiii iiii iiii iiii
SW	IFMT	1010 11rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1011 00rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1011 01rr rrrR RRRR iiii iiii iiii iiii
SF	IFMT	1011 10rr rrrR RRRR iiii iiii iiii iiii
SD	IFMT	1011 11rr rrrR RRRR iiii iiii iiii iiii
SEQUI	IFMT	1100 00rr rrrR RRRR iiii iiii iiii iiii
SNEUI	IFMT	1100 01rr rrrR RRRR iiii iiii iiii iiii
SLTUI	IFMT	1100 10rr rrrR RRRR iiii iiii iiii iiii
SGTUI	IFMT	1100 11rr rrrR RRRR iiii iiii iiii iiii
SLEUI	IFMT	1101 00rr rrrR RRRR iiii iiii iiii iiii
SGEUI	IFMT	1101 01rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1101 10rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1101 11rr rrrR RRRR iiii iiii iiii iiii

RES	IFMT	1110 00rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1110 01rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1110 10rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1110 11rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1111 00rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1111 01rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1111 10rr rrrR RRRR iiii iiii iiii iiii
RES	IFMT	1111 11rr rrrR RRRR iiii iiii iiii iiii

OPCODE LIST 2

INTEGER OPERATIONS AND OTHER OPCODES

SLLI	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 0000
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 0001
SRLI	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 0010
SRAI	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 0011
SLL	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 0100
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 0101
SRL	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 0110
SRA	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 0111

RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 1000
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 1001
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 1010
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 1011
TRAP	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 1100
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 1101
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 1110
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx00 1111
SEQU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 0000
SNEU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 0001
SLTU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 0010
SGTU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 0011
SLEU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 0100
SGEU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 0101
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 0110
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 0111
MULT	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 1000
MULTU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 1001
DIV	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 1010
DIVU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 1011
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 1100

RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 1101
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 1110
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx01 1111
ADD	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 0000
ADDU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 0001
SUB	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 0010
SUBU	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 0011
AND	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 0100
OR	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 0101
XOR	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 0110
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 0111
SEQ	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 1000
SNE	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 1001
SLT	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 1010
SGT	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 1011
SLE	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 1100
SGE	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 1101
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 1110
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx10 1111
MOVI2S	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 0000
MOVS2I	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 0001

MOVF	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 0010
MOVD	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 0011
MOVFP2I	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 0100
MOVI2FP	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 0101
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 0110
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 0111
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 1000
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 1001
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 1010
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 1011
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 1100
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 1101
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 1110
RES	RFMT	0000 00rr rrrR RRRR rrrr rxxx xx11 1111

OPCODE LIST 3

FLOATING POINT OPERATIONS

ADDF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 0000
SUBF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 0001
MULTF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 0010
DIVF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 0011
ADDD	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 0100

SUBD	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 0101
MULTD	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 0110
DIVD	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 0111
CVTF2D	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 1000
CVTF2I	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 1001
CVTD2F	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 1010
CVTD2I	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 1011
CVTI2F	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 1100
CVTI2D	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 1101
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 1110
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx00 1111
EQF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 0000
NEF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 0001
LTF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 0010
GTF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 0011
LEF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 0100
GEF	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 0101
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 0110
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 0111
EQD	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 1000

NED	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 1001
LTD	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 1010
GTD	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 1011
LED	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 1100
GED	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 1101
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 1110
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx01 1111
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 0000
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 0001
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 0010
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 0011
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 0100
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 0101
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 0110
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 0111
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 1000
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 1001
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 1010
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 1011
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 1100
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 1101
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 1110
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx10 1111

RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 0000
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 0001
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 0010
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 0011
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 0100
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 0101
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 0110
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 0111
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 1000
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 1001
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 1010
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 1011
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 1100
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 1101
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 1110
RES	RFMT	0000 01rr rrrR RRRR rrrr rxxx xx11 1111

Appendix B

The LOGIC III Code for the Variable Datapath Elements

The next three sections describe a 32-bit ripple carry adder, a 32-bit parallel prefix adder and finally a 32-bit barrel shifter in LOGIC III. These descriptions are generic and can be tailored to any size of the adder or shifter. However, the description of the last two functional units requires that the operand size be an exponent of 2.

B.1 Ripple Carry Adder: 32 bits

```
(*******************************************************)
(*******************************************************)
(*                                                     *)
(* The LOGIC-III code below defines a generic ripple   *)
(*carry  adder of any size N .                         *)
(*Circuit adder32 is a 32 bit instantiation of the     *)
(*ripple   carry  adder defined by the net module      *)
(*adder                                                *)
(*  A and B are 32bit wide primary                     *)
(*  inputs. cin is the carry in to the                 *)
(*  adder. S is the result of the                      *)
(*  addition and cout is the                           *)
(*  carry out result                                   *)
(*                                                     *)
(*******************************************************)
(*******************************************************)
GLOBAL
(* Include the OASIS standard cell definition libraries *)
includedef( "/usr/oasis/lib/scmos2.0/standard.def");
END.


(* LOGIC MODULE FA defines a Full Adder unit
which is used to build the ripplecarry
adder in the NET MODULE adder *)
LOGIC_MODULE FA(Cin,a,b:INPUT;sum,Cout:OUTPUT;);
 BEGIN if(~Cin)then
                begin
                    sum:=a XOR b;
```

```
                    Cout:=a AND b;
                end
            else
                begin
                    sum:=a XNOR b;
                    Cout:= a OR b;
                end;
 END.{LOGIC_MODULE FA}


NET_MODULE adder(N:INTEGER; A,B:array[0..N-1]of INPUT;
                 Cin: INPUT; Cout:OUTPUT; S:array[0..N-1]of OUTPUT;);
 VAR i:INTEGER;
     C:array[0..N] of NODE;
 BEGIN
     connect(C[0],Cin);
     for i:=0 to N-1 do
        FA(C[i],A[i],B[i],S[i],C[i+1]);
     connect(Cout,C[N]);
 END.{NET_MODULE adder}


(* Circuit adder32 is a 32 bit instantiation of the
NET MODULE adder declared above *)
CIRCUIT  adder32;
 VAR
      A,B:array[0..31]of input;
      cin:input;
```

```
    cout:output;
    S:array[0..31]of output;
 (* A and B are 32bit wide primary
 inputs. cin is the carry in to the
 adder. S is the result of the
 addition and cout is the
 carry out result *)
BEGIN
    adder(32,A,B,cin,cout,S);
END.{CIRCUIT  adder32}
```

B.2 Parallel Prefix Adder: 32 bits

```
(****************************************************)
(****************************************************)
(*                                                  *)
(* The LOGIC-III code below defines a generic parallel*)
(*prefix adder of a size which is an exponent of 2.  *)
(*Circuit adder32 is a 32 bit instantiation of the   *)
(*parallel prefix adder defined by the net module    *)
(*adder                                              *)
(*  A and B are 32bit wide primary                    *)
(*  inputs. cin is the carry in to the               *)
(*  adder. S is the result of the                    *)
(*  addition and cout is the                         *)
(*  carry out result                                 *)
(*                                                  *)
(****************************************************)
(****************************************************)

GLOBAL
includedef("/usr/oasis/lib/scmos2.0/standard.def");
END.

LOGIC_MODULE A_cell(a,b,c:input;g,p,s:output;);
BEGIN
     g:= a AND b;
     s:= a XOR b XOR c;
     p:= a OR b;

END.{LOGIC_MODULE A_cell}
```

```
LOGIC_MODULE B_cell(gj1k,pj1k,gij,pij:input;
                    gik,pik:output;
                    ci:input;cj1:output;);
{ ct last op par }
BEGIN
      pik:=pij AND pj1k;
      gik:=gj1k OR (pj1k AND gij);

      cj1:=(pij AND ci)OR gij;
      {ct:=ci;}
END. {LOGIC_MODULE B_cell}


LOGIC_MODULE carrygen(g,p,c:input;cout:output;);
BEGIN
    cout:=g OR ( p AND c);
END.{LOGIC_MODULE carrygen}


NET_MODULE adder(N:integer;A,B:array[0..N-1]of input;
                 S:array[0..N-1]of output;cin:input;
                 cout:output;);
VAR
     g,p,C:array[0..N-1] of node;
     G,P:array[0..N-1,0..N-1]of node;
     tmpr:node;
     j,i,k,l:integer;
BEGIN
```

```
connect(cin,C[0]);

for i:=0 to N-1 do
  A_cell(A[i],B[i],C[i],g[i],p[i],S[i]);

j:=N/2;
for i:=0 to j-1 do
   B_cell(g[i*2+1],p[i*2+1],g[i*2],p[i*2],
          G[i*2+1,i*2],P[i*2+1,i*2],C[i*2],C[i*2+1]);

j:=j/2;
k:=4;
while(j>1) do
BEGIN
  for i:=0 to j-1 do
    BEGIN
      B_cell( G[k-1+i*k,k/2+i*k] , P[k-1+i*k,k/2+i*k],
              G[k/2-1+i*k,i*k],P[k/2-1+i*k,i*k],
              G[k-1+i*k,i*k],P[k-1+i*k,i*k],
              C[i*k], C[i*k+k/2]);
    END;{for i:=0 to j-1 do}
k:=2*k;
j:=j/2;

END;{ while(j>1) do}

B_cell(G[N-1,N/2],P[N-1,N/2],G[N/2-1,0],
       P[N/2-1,0],G[N-1,0],P[N-1,0],C[0],C[N/2]);
{ the first 2 C[0s] are 0 really }
carrygen(G[N-1,0],P[N-1,0],cin,cout);
```

```
END.{NET_MODULE adder}


(*Circuit adder32 is a 32 bit instantiation of the
parallel prefix adder defined by the net module
adder *)
CIRCUIT adder32;
VAR
  (* A and B are 32bit wide primary
  inputs. cin is the carry in to the
  adder. S is the result of the
  addition and cout is the
  carry out result *)
  A,B:array[0..31]of input;
  S:array[0..31]of output;
  cin:input;
  cout:output;
BEGIN
adder(32,A,B,S,cin,cout);
END.
```

B.3 Barrel Shifter: 32 bits

```
(****************************************************)
(****************************************************)
(*                                                  *)
(* The LOGIC-III code below defines a generic ripple *)
(*barrel shifter of a size N which is a exponent of 2.*)
(*{shifter 32 is a 32-bit instantiation of the       *)
(*net module shifter which describes a barrel shifter.*)
(*The input to the shifter is "in" which is 32-bits  *)
(*wide and the output of the shifter is "out" which is*)
(*again 32-bits wide. "sh" contains the shift amount  *)
(*to be done for the  32 bit input.}                 *)
(*                                                  *)
(****************************************************)
(****************************************************)
GLOBAL
includedef( "/usr/oasis/lib/scmos2.0/standard.def") ;
END.

LOGIC_MODULE selector(a,b,s:INPUT;c:OUTPUT;);
 BEGIN if(s)then
              c:=b
             else
              c:=a;
 END.{LOGIC_MODULE selector}

NET_MODULE shifter(N,K:INTEGER; in:array[0..N-1]of INPUT;
                   out:array[0..N-1]of OUTPUT;
                   sh:array[0..K-1]of INPUT;);
VAR i,j,exp,jmod:INTEGER;
```

```
    p:array[0..K,0..N-1] of NODE;
BEGIN
     exp:=1;

     for i:=0 to K-1 do
       BEGIN
       for j:=0 to N-1 do
          BEGIN
          jmod:=(j+exp)-((j+exp)/N)*N;
          selector(p[i+1,j],p[i+1,jmod],sh[i],p[i,j]);
          END;{for j:=0 to N-1 do}
       exp:=exp*2;
       END;{for i:=0 to K-1 do}

     for j:=0 to N-1  do
       BEGIN
          connect(in[j],p[K,j]);
          connect(out[j],p[0,j]);
       END;{ for j:=0 to N-1  do }
END.{NET_MODULE shifter}


CIRCUIT  shifter32;
{shifter 32 is a 32-bit instantiation of the
net module shifter which describes a barrel shifter.
The input to the shifter is "in" which is 32-bits
wide and the output of the shifter is "out" which is
again 32-bits wide. "sh" contains the shift amount
to be done for the  32 bit input.}
 VAR
```

```
     in:array[0..31]of input;
     sh:array[0..4]of input;
     out:array[0..31]of output;
BEGIN
     shifter(32,5,in,out,sh);
END.{CIRCUIT  shifter32;}
```

BIBLIOGRAPHY

[Bri89] J. V. Briner, Jr. *LDVSIM: A Mixed LEvel System Simulator.* Technical Report 89-29, Duke University, Department of Computer Science, Durham, North Carolina, 1989.

[Cho89] Paul Chow *ed.* *The MIPS-X RISC Microprocessor.* Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[CH89] Paul Chow and Mark Horowitz. *The Design and Testing of MIPS-X* . Advanced Research in VLSI, Proceedings of the Fifth MIT Conference, March 1988. The MIT Press, Cambridge, Massachusetts, 1988.

[ERN+89] D. J. Erdman, D. J.Rose, G. B. Nifong, R. Subrahmanyam and S. Kenkel. *CAzM Circuit AnalyZer with Macromodelling,* Version Release 4.0 . Microelectronics Center of North Carolina and Duke University, July 1989.

[GM87] C. E. Gimarc and V. M. Milutinovic. A Survey of RISC Processors and Computers of the Mid-1980's. *IEEE Computer*, pages 59–69, September 1987.

[HJP+83] J. L. Hennessy, N. P. Jouppi, S. A. Przybylski, C. Rowen and T. R. Gross. Design of a High Performance VLSI Processor. *Proceedings of Third Caltech Conference on VLSI.* Computer Science Press, 1983.

[Hen84] J. L. Hennessy. *VLSI Processor Architecture. IEEE Transactions on Computers*, Vol. C-33, Pages 1221-1246, December 1984.

[HP90] J. L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufmann, 1990.

[HS90a] S. Ho and L. Snyder. Balance in Architectural Design. In *Proc. of the 17th International Symposium on Computer Architecture.* ACM, 1990.

[HS90b] T. J. Holman and L. Snyder. Architectural Tradeoffs in Parallel Computer Design. In *Proc. of the 1989 Decennial Caltech Conference: Advanced Research in VLSI*, pages 317–334. Caltech, MIT Press, 1990.

[HFY+83] R. J. Horning, M. Forsyth, J. Yetter, L. J. Thayer. How IC's impact workstations. In IEEE *Spectrum* , April 1991.

[Kat84] Manolis G. H. Katevenis. *Reduced Instruction Set Computer Achitectures for VLSI.* ACM Doctoral Dissertation Award 1984, MIT Press, 1984.

[KB88] G. Kedem and F. Brglez. OASIS: Open Architecture Silicon Implementation System. Technical Report MCNC TR 88-06, Microelectronics Center of North Carolina, February 1988. also appears in Proc IEEE International Symposium on Circuits and Systems, 1990 as *OASIS: A Silicon Compiler for Semicustom Design* by Kedem, Brglez and Kozminski.

[Ka88] G. Kane. *MIPS RISC Architecture: MIPS R2000/ MIPS R3000.* Prentice Hall, 1988.

[KF89] L. Kohn and S. Fu. A 1,000,000 Transistor Microprocessor . In *1989 IEEE International Solid-State Circuits Conference Digest of technical Papers.*

[Kun86] H.T. Kung. Memory Requirements for Balanced Computer Architectures. In *Proceedings of the 13th Annual Symposium on Computer Architecture.* IEEE, 1986.

[L.I87] L.I.S. VLSI Design Tools Reference Manual. Technical Report 87-02-01, Northwest Laboratory for Integrated Systems, University of Washington, Seattle, Washington, February 1987.

[Mok86] Nicolas Mokhoff. New RISC machines appear as hybrids with both RISC and CISC features. *Computer Design.* April 1, 1986.

[PT91] M. Pandey and A. Tyagi. RISC Microprocessor Implementations with Resource Allocation Balanced for Instruction Mix. Submitted for publication to ICCD, 1991.

[Pat85] D. Patterson. Reduced Instruction Set Computers. *Communications of ACM*, 28:8–21, January 1985.

[Per89] T. S. Perry. Intel's Secret is Out. *IEEE Spectrum*, pages 22–28, April 1989.

[PGH+84] S. A. Przybylski, T. R. Gross, J. L. Hennessy, N. P. Jouppi, and C. Rowen. Organization and VLSI Implementation of MIPS. *Journal of VLSI and Computer Systems*, 1(2):170–208, 1984.

[Ros90] R. Ross. Roger Ross on RISC. In *Computer Design* , September 1, 1990.

[SMH+86] W. S. Scott, R. N. Mayo, G. Hamachi and J. K. Ousterhout, eds. *1986 VLSI Tools: Still More Works by the Original Artists.* Report No. UCB/CSD 86/272, Computer Science Division (EECS), University of California, Berkeley, California, December 1985.

[SKP+84] R. W. Sherburne Jr., Manolis G. H. Katevenis, D. A. PAtterson, C. H. Sequin. A 32b NMOS Microprocessor with a Large Register File. *IEEE Journal of Solid-State Circuits, SC-19(5):682-689, October 1984.*

[Smi78] F. D. Smith. *Models of Multiprocessing for Transaction-Oriented Computer Systems.* PhD thesis, Dept. of Computer Science, University of North Carolina, Chapel Hill, October 1978.

[Sta90] W. Stallings. *Computer Organization and Architecture.* Macmillan Publishing Company, 1990.

[Sta89] R. M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., February 1989.

[Tab87] Daniel Tabak. *RISC Architecture.* Research Studies Press, 1987.

[Ter83] C. J. Terman. *Simulation Tools for Digital LSI Design.* PhD thesis, Department of Electrical Engineeing, M.I.T., Cambridge, MA, 1983.

[Tho90] J. Thompson. The embedded RISC war zone. *Electronic Engineering TIMES*, Issue 618, November 26, 1990.

[WE85] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addision-Wesley 1985.

[Wir91] L. Wirbel. HP takes RISC to end users. *Electronic Engineering TIMES*, page84, Issue 634, March 25, 1991.

[Wei91] R. Weiss. MIPS RISC headed for embedded control. *Electronic Engineering TIMES*, Issue 614, Jan 25, 1991.